CSC173
Lambda Calculus 2014

Please write your name on the bluebook. You may use two sides of handwritten notes. There are 90 possible points and 75 is a perfect score. Stay cool and please write neatly.

1. Lambda Calculus Evaluation (10 min)

Evaluate this expression (β-reduce with normal-order evaluation):

`(((λx. λy. λz.((x y) z) λf. λa.(f a)) λi.i) λj.j)`

Ans.

```
(((λx.λy.λz.((x y) z) λf.λa.(f a)) λi.i) λj.j) =>
((λy.λz.((λf.λa.(f a) y) z) λi.i) λj.j) =>
(λz.((λf.λa.(f a) λi.i) z) λj.j) =>
((λf.λa.(f a) λi.i) λj.j) =>
(λa.(λi.i a) λj.j) =>
(λi.i λj.j) =>
λj.j
```

2. Function Equivalence (10 min)

Show that functions a) and b) are equivalent by applying each to some abstract argument `<arg>`. Only expand names of functions when really necessary (just before they're applied, and then only if their effect is not obvious).

a) `identity`
b) `(self-apply (self-apply select-second))`

Ans.

a) `(identity <argument>) => ... =>`
`<argument>`

b) where `backslash la` means λ:

```
((self-apply (self-apply select-second)) <argument>) => ... =>
(((self-apply select-second) (self-apply select-second))
   <argument> => ... =>
(((select-second select-second) (select-second select-second))
   <argument>) => ... =>
((((\la f \la s.s select-second) (select-second select-second)))
       ((\la s.s (select-second select-second) <argument>) => ... =>
{((select-second select-second) <argument>) => ... =>
(\la s.s <argument>) =>
<argument>
```

or I'd accept a derivation based on the fact that select-second applied to anything is identity!

3. Logical Operators (20 min)

Recall we defined logical OR:
`def or = λx. λy.(((cond true) y) x),`
which simplified to
`def or = λx. λy.((x true) y).`

3.1 (5 min) : Write a C-language-like `x?y:z` conditional expression or a simple `if-then-else` statement that gives the boolean function NOR (that is, $\neg(P \vee Q)$). Only use the variables `x, y,` the operator `not,` and the constant `true` in your conditional expression.

Ans. x?(not true):(not y). OR, if x then (not true) else (not y)

3.2 (5 min) : Convert your expression for NOR into a λ-calculus `cond` expression: it should have two λs, one `cond`, and however many `true, not, x, y`s you need.

One Ans. `λx.λy. (((cond (not true)) (not y)) x).`

3.3 (5 min) : Evaluate and simplify (you'll remove *cond* from) the inner body of this expression to get a `λx. λy. (simple, i.e. no-cond, expression)` for NOR.

One Ans. `def nor = λx. λy. ((x (not true)) (not y))`

Whose inner body came from...

```
(((cond (not true))(not y)) x) =>
(((λe. λf. la c.((c e) f) (not true)) (not y)) x) =>
((λf. λc. (( c (not true)) f ) (not y)) x) =>
(λc. (( c (not true)) (not y) x) =>
((x (not true) (not y))
```

3.4 (5 min): Work out cases needed to prove a version of DeMorgan's law by model checking:
To Prove:
$\neg(\neg A \vee \neg B) = (A \wedge B)$.

Using `nor`, that's `NOR (( NOT X) (NOT Y)) = AND (X Y)`. For each case below apply your simplified expression for NOR from 3.3 to the two arguments and show it evaluates to the correct answer. Since the expression will involve strictly `true, false, not`, you can rewrite to use only `true, false`, which are `select-first, select-second`, and so do each case in one line. No need to expand to λ-level.

Here's the first case for free: Let (X,Y) have values (T, T), so (skipping an obvious step) args to NOR are (F,F):

```
(((not true) (not true)) (not (not true))) ==>
((false (not true)) (not false)) ==>
(false false true) ==> true,
```

which is right!

Now you do the three cases for (T,F), (F,T), and (F,F)

Ans.

TF→FT `((false (not true)) false) ==>`

```
(false false false) ==> false
```

FT→TF ((true (not true)) false) ==>
(true false false) ==> false

FF→TT ((true (not true)) false) ==>
(true false false) ==> false

Thus, indeed NOR ((NOT X) (NOT Y)) = AND (X Y).

4. Recursion (15 min)

Given: Y = λf.(λx. f(x x)) (λx. f(x x))
and an abstracted high-level definition of factorial:
fact = λf.λn. (if n = 0 then 1 else n * (f (n-1))).

Now compute factorial of 2 recursively using Y. You don't need to expand any operators except Y and fact (as above), and infix notation is OK (e.g. 2*3 would reduce to 6). I'll get you started (and finished)

(Y fact) 2 =>
=>...=> (hint: first thing to do is rewrite Y using its definition)
2

Ans.

```
(Y fact) 2 => // replacing Y w/ encoding
(λf.(λx.f (x x)) (λx.f (x x)) fact) 2 => // beta-reduction: 1st f => fact
(λx. fact(x x)) (λx.fact (x x)) 2 => // beta-reduction: 1st x => λx.fact (x x)
= (fact ((λx. fact(x x)) (λx.fact (x x)))) 2 =>
// apply encoding for (Y fact)
// ((λx. fact (x x)) (λx.fact (x x))) => (Y fact) =>
// we know this is the encoding for (Y fact) from 3rd line of work so far
(fact (Y fact)) 2 => // apply encoding for fact
(λf.λn. if n = 0 then 1 else n * (f (n-1)) (Y fact) 2 =>
// -reduction: 1st f -> (Y fact)
(λn. if n = 0 then 1 else n * ((Y fact) (n-1))) 2 => // beta-reduction: n -> 2
if 2=0 then 1 else 2 * ((Y fact) (2-1)) => // apply if
2 * ((Y fact) 1) =>...=> // evaluate if
2 * 1 => // apply *
2
```

5. Number Representations (15 min)

5A. (10 min)

Recall Scott's number encoding, which is best understood with "applicative semantics", meaning evaluate arguments before applying functions. In particular (using compact notation for 2-argument functions)

zero = λxy.y (i.e. select-second)
2 = λxy.x(λxy.x(λxy.y)) (i.e. sel-1st sel-1st sel-2nd).

We saw that we can write the predecessor function for Scott as pred = λz.z (λp.p) 0.

Why? `pred N` first copies N to the front with first identity function λz.z. Now N is applied to the last two arguments: N's first λ- expression is `sel-1st`, which chooses arg1 (λp.p), and ignores arg2 (0). The identity function (λp.p) is thus substituted in place of N's first `sel-1st`'s body (so the first `sel-1st` vanishes), and applied to the rest of the original N, yielding the representation for N-1. If N=0 (i.e. sel-2nd), copying it up front and applying it to the two arguments as before selects arg2 (0).

Now, using `pred` above as a template, give a similar-looking one-line implementation of `iszero` for Scott encoding. (Hint: use `true, false`.)

5B. (5 min) We wonder: "Why do all three of our number representations basically use `select-second` for `succ`?" Could we just swap the roles of `select-first, select-second` and get three new mirror-image number representations?

Ans. (from on-line lectures)

5A:

`iszero = λz.z (λx. false) true`

5B.: No reason, should work fine unless I'm missing something: in fact I saw a version of Scott's encoding with this swap.

6. Continuations I (10 min)

What gets displayed here?

```
(let ((start #f))

  (if (not start)
      (call/cc (lambda (cc)
                 (set! start cc))))

  (display "Going to invoke (start)\n")

  (start))
```

Ans.

```
Going to invoke (start)
Going to invoke (start)...%[forever]
```

7. Continuations II (10 min)

An alternative interface for interacting with continuations consists of two procedures: Let's call them `right-now` and `go-when` and define them and some code:

```
(define (right-now)
  (call-with-current-continuation
   (lambda (cc)
     (cc cc))))
```

```
(define (go-when then)
  (then then))
;;----------------
(define when #f)

(define demox
  (lambda (n)
    (cond
      ((< n 4) (set! when (right-now))
               (display  "small")
               (newline))
      (else    (set! when (right-now))
               (display "bye")
               (newline)))))
```

A. (8 min) What gets printed if we now type the following, in order, at the listener?

`(demox 3)`:

`(demox 5)`:

`(go-when when)`:

B. (2 min). Describe in words what `right-now` and `go-when` do.

Ans:

```
> (demox 3)
small
> (demox 5)
bye
> (go-when when)
bye
```

Ans:

`right-now` returns the current point (moment) in the computation.

`go-when` jumps to some point (moment) returned by right-now.